Bcfg2: A Pay As You Go Approach to Configuration Complexity

Narayan Desai

December 5, 2005

Abstract

While configuration management tools are an area of substantial research and development, tool adoption has lagged behind. We assert that lack of adoption is caused in large part by complexity costs. We will describe bcfg2, a configuration management tool, and its approach to complexity mitigation.

1 Introduction

Configuration management is a topic that is intimately associated with system management, security, and fundamentally, computer usage itself. As the ubiquity of computers grows, the raw number of computers which need proper configuration and software upgrades grows as well. System administrators have great familiarity with the issues of configuration management; most environments have a regular cycle of software patching and reconfiguration that must be performed.

While this task can be onerous, it is absolutely essential; without these regular configuration changes, systems can easily become insecure or otherwise non-functional. As tools can make this process faster and less error prone, the deployment of configuration management tools can dramatically improve the reliability and security of computer networks.

While it is generally acknowledged that configuration management tools can greatly improve the process of system administration, widespread adoption has not generally occurred. This is the most serious problem in configuration management. Bcfg2 is by no means alone as a comprehensive option; LCFG and Quattor are both comparable. However, none of these tools has seen any serious adoption outside of their development environment yet.

Our experiences deploying Bcfg2 have suggested that complexity is a major aspect of the adoption puzzle. All of the systems mentioned above are quite powerful, however, the price of this power is complexity. System complexity causes a number of issues. These tools can be foreign or counter-intuitive to users unfamiliar with them. Basic administration mechanisms change substantially with each of these tools. We have found that bcfg2 users generally start

with simpler configuration representations. As their familiarity with the tool increases, users grow more willing to accept more complex representations in order to gain better functions from the tool.

Bcfg2 attempts to mitigate complexity issues through the use of generators. Bcfg2 uses generators as configuration sources, each of which can have different representations of configuration data. Several generators can be used in parallel, each contributing different aspects of the configuration, and each using a different configuration representation. This feature allows Bcfg2 to provide a "pay as you go" approach to configuration representation complexity. Users can start with simple generators that use a very literal representation of configuration data. As users grow more familiar with Bcfg2, and grow to trust it more, they can shift their configuration data to more sophisticated representations that allow more reuse and recomposition.

We will discuss several generators and the varieties of configuration representations they use. In each case, we will describe the generator's function and benefits and disadvantages to the configuration format's use. We will also attempt to provide a sense of where each fits into the long term adoption process.

2 Related Work

Several configuration management tools comparable to Bcfg2 exist. LCFG[2] and Quattor[6] are most similar to in functionality to Bcfg2. LCFG and Quattor both use parameterized configurations as a basis for system configurations. That is, both of these tools constructs a parameter list per client. Parameters consists of keys and associated values. These parameter lists are interpreted by software on clients, resulting in configuration changes. Quattor was designed by a group with substantial LCFG experience, so it has a similar overall model. The parameter store in Quattor consists of a database, where LCFG uses flat files and a preprocessing scheme for client parameter construction.

CFEngine[4], while being the prevalent configuration management tool, is vastly unlike Bcfg2. CFEngine is functionally a domain-specific programming language, tailored to system administration. It is designed around an imperative model; users write scripts that alter client configurations into the desired state. On the other hand, Bcfg2, LCFG, and Quattor use a declarative model. The user produces a specification for client configurations. The tool determines how to best reconfigure clients in light of this specification.

Much work has been done in the area of configuration tool usability and deployment studies. Rémy Evard performed a survey of system configuration techniques in 1997 [10]. Paul Anderson has performed a more recent survey as a part of the GridWeaver project [1]. These surveys have found large numbers of tools used for system configuration, many of which are site-specific. This finding highlights the adoption problem, as it demonstrates the lack of external adoption of system configuration tools.

More recently, we have performed a case study in system configuration tool deployment in a large group [9]. The deployment process, conducted over a

period of months, highlighted the difficulty inherent in the adoption of a system configuration tool. Also, a cost analysis of system configuration and administration has been conducted. This found a number of interesting issues focused around the methods used in system configuration tool use.[7]

All of this work points at tool adoption as the most serious problem in system configuration at this point. Fundamentally, tools will only help the community at large if they can be easily used.

3 Tool Adoption

The adoption of a system configuration tool can be a costly and disruptive process. This high cost is caused by several factors. System configuration tools implement functionality that is intimately entangled with the daily activities of system administrators. Due to this relationship, tool adoption (of any tool) causes large changes to daily administration tasks.

A central task during this adoption process is the creation of a configuration specification. Different tools use different mechanisms to represent this specification. LCFG and Quattor use parameterized specifications. This mechanism is quite powerful, however, initial setup is quite time consuming. Several other systems, including Bcfg2, use more literal representations of configuration, including copies of configuration files. Bcfg2 also uses other representations as well.

We have completed several deployments of Bcfg2, among several different groups of administrators. We have found the adoption process to be fraught with other issues as well. Different administrators will have different considerations in tool selection. This is largely due to differences in prior experience. As each administrator discovers appropriate ways to use the new tool, they find disparate, and, in some cases, conflicting mechanisms to achieve the same goals. This issue is the most troubling problem caused by expressive, powerful tools. Different mechanisms are developed for many reasons; administrators often have different goals, tasks and experiences. Also, as administrators become more proficient with a given tool, their usage (and comfort level with tool related complexity) increases. These issues are described in much greater detail in another paper[9]. Suffice it to say that this process can be quite intense and costly.

4 Bcfg2

Bcfg2[8] was originally designed as a system configuration tool research vehicle. Over the last three years, the basic design principles have proven themselves in several environments. A major revision was completed in the last year, resulting in a production-quality tool. Bcfg2 is now deployed in several environments, and has been publicly released [3].

4.1 Architecture

Bcfg2 is designed as a client/server application. The client is responsible for fetching a configuration description from the server, inventorying local client state and committing appropriate configuration changes. The server is responsible for marshalling the overall configuration specification into a client-specific configuration description.

4.1.1 Client

The Bcfg2 client is run as a normal program on each client. The client configuration process consists of three steps. It connects to the server, and fetches a series of probes. These probes are executed, and the results are uploaded to the server. Probes detect local contributions to system configuration and can be used to discover hardware inventories or any other locally canonical data.

After the probe interaction, the client fetches a literal configuration description from the server. This description consists of dependent and independent clauses of configuration entries. Entries can have one of five basic types: ConfigFile, Package, Service, Symlink, and Directory. Dependent clauses, or bundles, contain a set of entries that have installation-time interdependencies. Bundles generally correspond to services, their configuration files, and associated software packages.

Once the client has downloaded the configuration description, it inventories itself. It compares this with all configuration entries in the description. Any nonconforming entries are flagged for correction. It then checks for configuration entries that aren't included in the description. This process is called two-way verification. It provides fairly rigorous configuration verification.

Once the inventory step has completed, the client corrects any misconfigurations. This process loops correcting errors until no more progress can be made. It also provides automatic dependency handling. At the end of this stage, all reconfiguration that can be made has been done. That is, with no changes, an additional client execution will not result in any additional actions.

Upon update completion, the client produces a message containing a number of statistics including overall state, failing configuration entries, modified configuration entries, and extra configuration entries. This message is sent to the server where it can be used to create system summaries.

In our experience, client details only impact adoption insofar as client reliability and result exposure are concerned. Most all complicated adoption issues occur as a result of server-side design decisions.

4.1.2 Server

The Bcfg2 server is responsible for rendering the overall configuration specification into a per-client configuration description, managing client settings (such as base image, profile, and other metadata), and maintaining client statistics. The system is designed around a central specification. Administrators describe the intended goal configuration in this specification. Clients start in some state,

not necessarily identical to the goal state. As client nodes run Bcfg2, discrepancies between clients and their desired configuration are individually corrected. These changes and the goal states are tracked in the client statistics. Through the use of both the specification and client statistics, three important pieces of information can be derived: the desired state, non-conforming clients, and a list of ways those clients are misconfigured. Using this information, administrators can determine all client states using only server-side information at any time.

The role played by the server is obvious during all client configuration steps, with the exception of the client configuration description creation process. This process consists of the following steps. First, the server locates metadata for the client requesting a configuration. This data is used to construct the abstract configuration. The abstract configuration is a skeleton configuration that contains an inventory of all configuration entries needed for the client configuration, but without any literal contents. For example, "ConfigFile" entries in the abstract configuration will have a filename but no file contents or permission information associated this it. The abstract configuration contains all of the entries that will appear in the complete, literal configuration. This step is used to map out bundles and create structure in the abstract configuration.

Once the abstract configuration has been constructed, literal data must be bound into each entry. The Bcfg2 server uses a set of generators for this task. Generators are server side loadable modules. They are enabled upon server startup, and register ability to provide literal configuration information about particular configuration entries. For example, a generator may register the ability to provide literal information for the configuration file "/etc/passwd". When the server needs to bind data into a configuration element, it locates the generator that can service for that element, and calls it. The generator can then execute arbitrary logic to determine the correct literal values for this entry on the given client.

The use of generators in the Bcfg2 server creates a number of interesting features. Most importantly, arbitrary representations can be used as specifications. This makes Bcfg2 an ideal testbed for specification languages. Second, arbitrary logic can be employed in the process of literal configuration construction. This means that any scheme imaginable can be used to construct configuration from a specification. Randomness and a variety of other programmatic techniques can be employed in this process. More importantly, actions can be associated with the configuration process.

A number of issues in the adoption process are centered around specification language. In the case of Bcfg2, we have found it useful to have multiple ways to specify similar configurations. This provides multiple models that can be used for different administrators as appropriate.

In general, administrators' trust in system configuration tools only grows with experience. Initially, administrators neither trust or have a deep understanding of any new tool. Common sense suggests that simple configurations will cause less problems than complex ones, so new users tend to create literal and straightforward configuration specifications.

The model provided by Bcfg2 allows administrators to work with a spec-

ification language as literal or abstract as required by the problem and the administrator's familiarity and comfort level.

5 Generators and Specification Languages

Due to the use of several different generators in the same Bcfg2 deployment, different languages with vastly different semantics are frequently used concurrently. We will describe several generators, and their associated specification languages. This will provide some examples of how Bcfg2 can use literal and symbolic specifications.

The use of client metadata is prevalent throughout all generators. The metadata for a client consists of its hostname, image, profile, set of classes, set of bundles, and set of attributes. In general, specification fragments will apply to some subset of clients based on one of these characteristics.

Each generator consists of a loadable module and a filesystem repository. The repository contains generator specific configuration specification.

5.1 Cfg

Cfg is a generator that implements a literal configuration file repository. For each configuration file served, a relative path in the filesystem repository is created. For example, for a configuration file like `/etc/X11/xorg.conf`, a directory corresponding to the path is created, in this case `./etc/X11/xorg.conf/`. Literal copies of configuration files are placed in this directory. Each file in this directory applies to some subset of clients, based on metadata. Applicability is determined based on the filename. All filenames start with the basename of the configuration file. If no suffix is specified, then the file applies globally, at the lowest priority. More specific files will have a suffix that contains a metadata type (like hostname, class, bundle, or image), the name of a group of that type, and a priority used in case of ties.

This generator provides a very literal and predictable representation of configuration. What you see is precisely what you get. This makes this generator one of the default set of generators enabled in Bcfg2 installations.

5.2 SSHbase

SSHbase is a ssh key management system. It maintains a repository of ssh public and private keys and constructs a comprehensive `ssh_known_hosts` file. It functions entirely automatically; new keys are generated when one is requested for an unknown host. At this point, the `ssh_known_hosts` file is updated for all clients.

The repository language used by this generator consists of a directory containing public and private keys for all clients. This system ensures two basic properties. First, client keys persist beyond client rebuilds. Second, client keys can be centrally revoked, in case of a system compromise.

The files in this directory are structured like host specific files in the `Cfg` repository. Each file consists of the key file basename followed by ".H_" and the client's hostname.

5.3 TCheetah

TCheetah is a generator based on the Cheetah templating language[5]. Configuration files are generated by instantiating a template populated with values from an XML document. These XML documents are contained in the TCheetah repository and contain two basic types: a keyval type, and a list type. List types can contain other list types or keyvals.

The TCheetah generator is quite powerful, particularly when compared with the previous two generators described. At the same time, this generator is much more complex, so its use typically occurs after administrators are comfortable with Bcfg2 overall.

In our environment, we used Bcfg2 for nearly a year before starting to use TCheetah. It is used to automatically build DHCP, DNS, and NIS files from the same canonical host data. Our current system could easily be extended to interface with LDAP systems as well.

This generator provides a powerful model that can be used to generate configuration files in multiple formats based on the same canonical data. This scheme is very similar to ones provided by LCFG and Quattor, and has proven itself quite capable.

5.4 ServiceMgr

The service manager generator provides service activation and deactivation information for clients. Its specification consists of a single XML file that contains metadata class based service information. This generator is a good example of a group of several generators that use a metadata class scoped XML file for configuration specification.

5.5 Other Generators

Several other generators exist and are in development. These provide several other capabilities and specification representations. Of the more complex ones, many could alternatively be implemented as a set of TCheetah templates. User interface issues are generally the motivating factor for alternative powerful generators. We have found that our administrators like domain specific languages in some cases. We have added a domain specific language for account and user access information, and one for webserver virtual host layout. Generators are easy to write, and can be added whenever it appears that another configuration representation may be useful.

Similarly, generators have a variety of other uses as well. They are well suited to tracking external data in local configuration files, as in the case of

tracking changes made globally to DNS. Generators are also useful for encoding administrative processes.

6 Results

The ability to choose (or indeed create) representations for configuration specification based on the individual situation has yielded a great number of benefits both during and after the adoption process. A number of tradeoffs are implicit in this decision. Depending on the situation, and the administrator's comfort level with the tool, a simple solution may be the most desirable. As time goes on, it may be desirable to migrate to a more complicated representation that provides more functionality.

This feature is provides a substantial departure from other system configuration tools. Bcfg2 provides the ability to migrate from a simple representation to a more sophisticated one on a configuration file by file basis. Usually, tools provide a single mechanism for specification. The ability to use many representations allows administrators to choose the deliberately take on cost where they think it will be valuable.

During our deployments, we have found that this ability makes administrators far more comfortable with the tool. In each case, administrators started with a simple, literal representation of their configuration. As time goes on, administrators are willing to assume more complexity in particular areas where there configuration needs are complicated.

We have found that these areas differ radically from environment to environment. In our cluster environments, our hardware configuration is static; hence, we are able to use a literal configuration with no problems. In our workstation environment, hardware is largely varied. This hardware also changes quite frequently. For this reason, we have adopted a more complicated system that automatically tracks client's hardware inventory and generates a proper set of configuration files for that hardware. This system is completely unnecessary for a largely static system like our clusters, and takes more time to maintain and tune. It is also harder to modify or explain to new administrators.

Conversely, user access to nodes in our clusters change frequently, so we are willing to accept more complexity in the system that manages system authentication data. Users are only granted access on nodes where their jobs are currently running. On our workstation environment, user access changes only when new users are added or old accounts are removed.

System administration is rife with these sorts of decisions. Being able to decide when it is worthwhile to assume additional complexity leads to systems that are as simple as is reasonable for a given environment.

7 Conclusion and Future Work

Tool complexity is a key issue preventing widespread deployment of system configuration tools. While many capable tools exist, the costs of deployment are too high for many environments to bear. Our approach of allowing representations of varying complexity has been demonstrated to improve the situation, particularly during the adoption process.

This technique has also provided a good migration path for our administration after initial deployment. It provides a good mechanism for administrators to tackle their complicated system configuration problems without incurring overhead where none is needed.

That said, much work remains to be done. Tool adoption is still much too difficult for successful widespread adoption. The adoption process could be vastly improved. Automatic specification acquisition would go a long way in helping administrators to try tools out. Better incremental adoption techniques would allow slow migration of environments onto system configuration tools.

Motivation is another issue that must be tackled. It is imperative that better system configuration techniques be adopted. However, short term compelling reasons are needed to justify up-front adoption costs. Tools that can be adopted incrementally will help with this issue; smaller scale initial deployments can begin providing benefits for parts of the environment before a complete specification is created. This also has the effect of lowering the cost of entry for these tools.

Bcfg2 will continue to be a testing ground for ideas relating to system configuration. We plan to add a facility for automatic change detection and integration. This will allow easy maintenance of already configured systems, and will ease the reconfiguration process. We also plan to create a tool that will create an initial configuration specification. This will help new users to try out Bcfg2. At our site, we plan to deploy Bcfg2 in a number of new environments. This will undoubtedly provide valuable information about adoption issues faced by new users.

References

[1] Paul Anderson, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, and Peter Toft. Technologies for large-scale configuration management. `http://www.gridweaver.org/WP1/report1.pdf`, 2002.

[2] Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, pages 363–372, Berkeley, CA, USA, 2000. USENIX.

[3] Bcfg2 web site. `http://www.mcs.anl.gov/cobalt/bcfg2/`.

[4] Mark Burgess. A site configuration engine. In USENIX, editor, *Computing Systems, Summer, 1995.*, volume 8, pages 309–337, Berkeley, CA, USA, Summer 1995. USENIX.

[5] Cheetah templating web site. `http://www.cheetahtemplate.org/`.

[6] Lionel Cons and Piotr Pozanski. Pan: A high-level configuration language. In *Proceedings of LISA 2002: 16th Systems Administration Conference*, pages 83–98. USENIX, 2002.

[7] Alva Couch. What is this thing called system configuration? `http://www.usenix.org/publications/library/proceedings/lisa04/tech/talks/couch.pdf`. Invited Talk LISA 2004.

[8] N. Desai, R. Bradshaw, R. Evard, and A. Lusk. Bcfg : a configuration management tool for heterogeneous environments. In *Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER03)*, pages 500–503. IEEE Computer Society, 2003.

[9] Narayan Desai, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Rémy Evard, Cory Luenighoener, Ti Leggett, J.P. Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey. A case study in configuration management tool deployment. In Usenix, editor, *Proceedings of the Nineteenth System Administration Conference (LISA XIX), December 4–9, 2005, San Diego, CA, USA*, pages 39–46, 2005.

[10] Rémy Evard. An analysis of UNIX system configuration. In USENIX, editor, *Proceedings of the Eleventh Systems Administration Conference (LISA XI), October 26–31, 1997, San Diego, CA, USA*, pages ??–??, Berkeley, CA, USA, 1997. USENIX.